JSF libraries en custom componenten

Heeft Java dan eindelijk zijn Oracle Forms?

In een serie van drie artikelen geeft Lucas Jellema een rondleiding door de wereld van Java Server Faces. De eerste twee delen boden een introductie op de basiselementen binnen de JSF-standaard en het inzetten van de ADF Faces implementatie van Oracle. Deze afsluitende aflevering bespreekt het combineren van JSF componenten uit verschillende libraries zoals Apache MyFaces en ICE-Faces en de ontwikkeling van eigen JSF-componenten. Als IDE maken we gebruik van Oracle JDeveloper 10.1.3.2. Hoewel enige Java-achtergrond nuttig is, zijn de artikelen toegankelijk voor alle Oracle ontwikkelaars.

Een beetje Googlen op internet levert al snel een omvangrijke lijst van verschillende open source JSF implementaties en libraries op. Hierbij is het verstandig onderscheid te maken tussen min of meer volledige implementaties die onderling niet (goed) gecombineerd kunnen worden enerzijds – zoals Trinidad, ICEFaces, ADF Faces, RCFaces en Project Woodstock – en componenten library's die wel aanvullend op elkaar of volledige implementaties kunnen worden ingezet – zoals Sun RI, Apache Myfaces Tomahawk en Tobago, JBoss RichFaces en de JSF Chart Creator. Naast dit open source aanbod zijn er diverse commerciële producten: ECruiser Suite, QuipuKit, NetAdvantage, WebGalileo Faces en Backbase JSF Edition. Zie voor een uitvoerig overzicht: http://www.jsfmatrix.net/.

Bij het onderzoek naar de aangeboden JSF-bibliotheken en -componenten vielen mij de integratiemogelijkheden onderling nogal tegen. Het blijkt erg belangrijk te zijn om een primaire JSF-implementatie te kiezen, waar je daarnaast wellicht nog enkele specifieke componenten aan toevoegt, maar die toch negentig procent van je behoefte invult. ADF Faces, MyFaces Trinidad en Project Woodstock (de stand-alone variant van het Sun Visual WebPack voor NetBeans) zijn goede opties voor deze basisset. Ze kunnen onderling niet worden gecombineerd, bijvoorbeeld omdat ze specifieke eisen stellen aan de render-kit of de afhandeling van AJAX-requests.

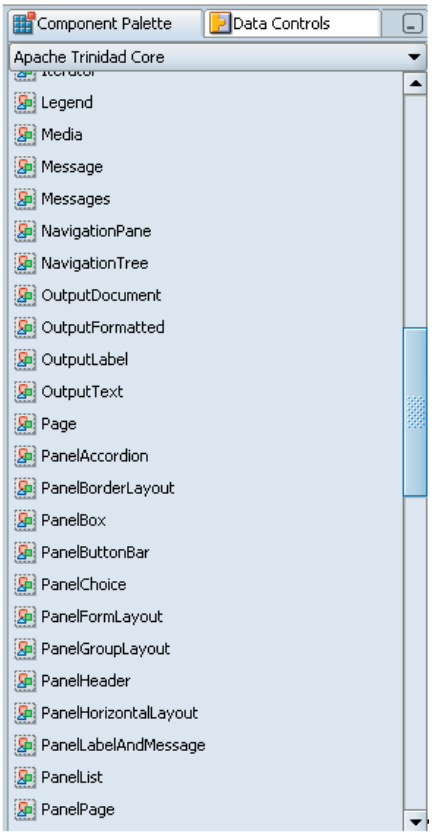
Deze implementaties kunnen over het algemeen wel worden gecomplementeerd met losse componenten zoals die bijvoorbeeld in MyFaces Tomahawk of JSF-Comp beschikbaar zijn. Later in dit artikel zal ik beschrijven hoe we een Tomahawk component gebruiken in een ADF Faces applicatie.

Apache MyFaces Trinidad

Na alle aandacht voor ADF Faces in de vorige aflevering van deze artikelenserie, zullen we nu eens kijken naar de ontwikkeling met JDeveloper van een JSF webapplicatie op basis van de open source tegenhanger: Apache MyFaces project Trinidad. Uiteraard zijn de overeenkomsten tussen ADF Faces en Trinidad groot. Toch zijn er ook verschillen: Trinidad heeft een aantal componenten die niet in ADF Faces zitten, zoals Chart, ChooseDate (geen popup window), InputNumberSpinbox en OutputDocument. Daarnaast kent Trinidad een eleganter menusysteem en lijken er ook andere verbeteringen geboren uit ervaring met ADF Faces te zijn doorgevoerd. Op termijn vinden deze verbeteringen ook vast hun weg weer terug naar ADF Faces, maar daarin zijn ze nu nog niet beschikbaar. Zoals gesteld, ADF Faces en Trinidad kunnen, zo laat Oracle Product Management desgevraagd en een tikje gegeneerd weten, "op dit moment *niet* gecombineerd worden binnen een webapplicatie." "Voor JDeveloper 11g zou het kunnen dat interoperabiliteit van ADF Faces met andere JSF implementaties een doelstelling wordt." werd er uiterst voorzichtig aan toegevoegd.

De allersnelste manier om met Trinidad aan de slag te gaan, is door naar http://people.apache.org/repo/m2-snapshot-repository/org/apache/myfaces/trinidad/trinidad-demo/1.0.1-incubating-SNAPSHOT/ te gaan en daar de file trinidad-demo-1.0.1-incubating-SNAPSHOT.war te downloaden. Maak vervolgens in JDeveloper een nieuwe applicatie aan, zonder technology scope of default project. Creëer dan een nieuw project, gebaseerd op een WAR file en selecteer de zojuist gedownloade trinidad-demo-1.0.1-incubating-SNAPSHOT.war . JDeveloper maakt een nieuw project aan met daarin alle benodigde libraries en bovendien een demo van alle Trinidad componenten. Je kunt de demo starten door componentDemos.jspx te starten.

Zelf een pagina ontwikkelen met Trinidad-componenten ver-



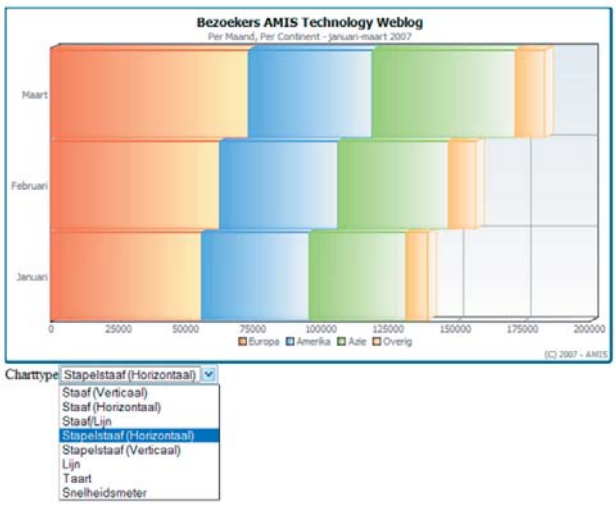
Afbeelding 1. Een deel van het Trinidad Component Palet.

loopt identiek aan de ontwikkeling met ADF Faces: een kwestie van componenten naar de pagina slepen, configureren, associëren met managed beans en eventueel Ajax (Partial Page Rendering) enablen.

Laten we een pagina ontwikkelen die gebruikmaakt van de Trinidad Chart component. Creëer een nieuwe JSF pagina. Sleep de Chart component van het palet naar de pagina. Sleep ook een SelectOneChoice naar de pagina. Deze laatste component gebruiken we om de gebruiker tussen verschillende grafiek-soorten te laten kiezen. De Trinidad Chart component ondersteunt een stuk of 16 verschillende types, van Taart en Staafdiagram tot Radar en Snelheidsmeter. Uiteraard kunnen we daarnaast allerlei extra componenten gebruiken om data of layout elementen of buttons te tonen. Configureer een bean die een ChartModel oplevert. Dit is een door Trinidad gedefinieerde class waar we de dataset langs x en y-as(sen) vastleggen en die zaken als titel en legenda bevat.

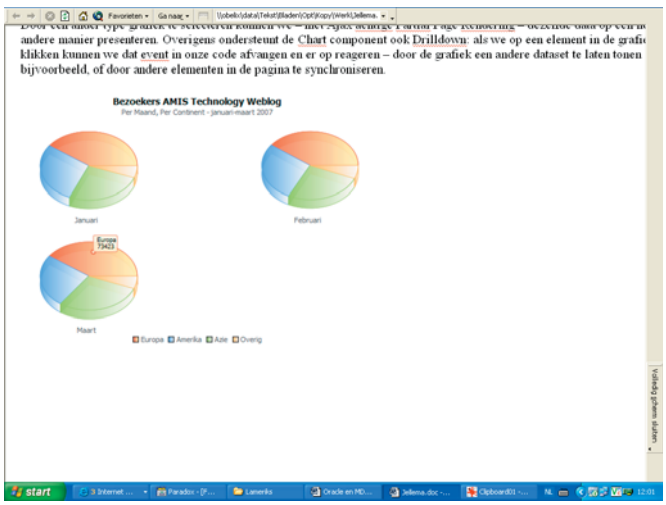
De code in de JSF pagina om de Trinidad Chart te gebruiken is:

```
<tr:chart value="#{MyChartBean.value}" id="chart"
          YMajorGridLineCount="7"
          inlineStyle="width:680px; height:400px;"
          partialTriggers="selectChartType"
          type="#{MyChartBean.chartType}"/>
```



Afbeelding 2. ChartModel, een door Trinidad gedefinieerde class.

De pagina ziet er dan – met grafiek - uit als in afbeelding 2. Door een ander type grafiek te selecteren kunnen we – met Ajax achtige Partial Page Rendering – dezelfde data op een heel andere manier presenteren. Overigens ondersteunt de Chart component ook Drilldown: als we op een element in de grafiek klikken kunnen we dat event in onze code afvangen en er op reageren – door de grafiek een andere dataset te laten tonen bijvoorbeeld, of door andere elementen in de pagina te synchroniseren.



Afbeelding 3. De Chart-component ondersteunt ook Drilldown.

MyFaces Tomahawk Componenten

Hoewel er met onderling compatibiliteit van JSF implementaties de nodige problemen blijken te bestaan, is de Apache MyFaces Tomahawk library een plezierige uitzondering. De Tomahawk componenten zijn vrij eenvoudig toe te passen in JSF applicaties met een andere primaire JSF-implementatie als fundament.

Details over Tomahawk zijn te vinden op: http://myfaces.apache.org/tomahawk/. Hier vind je ondermeer een overzicht van alle componenten en hoe ze te gebruiken. De lijst bevat momenteel een kleine dertig componenten, waaronder bijvoorbeeld een Calendar, Tree, Dropdown Menu, Rich HTML Editor, Scheduler en een Popup component. Deze laatste kan je gebruiken om een deel van de inhoud van een pagina te tonen in een popup – floating DIV – die verschijnt na een mouse-event. Ter demonstratie kijken we naar het toepassen van Tomahawk om een ADF Faces applicatie een extra table-overflow modus te geven: Style Popup. De applicatie zoals we die met ADF Faces en JHeadstart hebben ontwikkeld ziet er uit als in afbeelding 4.

Afbeelding 4. De applicatie zoals die met ADF Faces en JHeadstart is ontwikkeld.

De overflow die nu rechts naast de tabel staat en half van de pagina afvalt, wil ik graag tonen in een popup die wordt opgeroepen als de gebruiker een details-icon activeert.

De stappen zijn:

- Tomahawk downloaden (http://myfaces.apache.org/download.html) en opnemen in het project (extract tomahawk-1.1.3.jar uit de zip-file en kopieer naar WEB-INF/lib van de JSF applicatie waarin je Tomahawk componenten wilt gebruiken). Optioneel: voor sommige componenten moet de Tomahwak Extension Filter worden geconfigureerd in de web.xml file. Zie bijvoorbeeld http://technology.amis.nl/blog/?p=1336 voor details.
- De Tomahawk Taglibrary refereren in de betreffende JSF pagina's. Het Component Palet bevat nu de Tomahawk Componenten
- De Tomahawk Popup component als wrapper om de overflow PanelGroup plaatsen:

```
<t:popup styleClass="af_table_control-bar-top"
         closePopupOnExitingElement="false"
         closePopupOnExitingPopup="true"
         displayAtDistanceX="60" displayAtDistanceY="-150">
  <h:outputText value="Details"/>
  <f:facet name="popup">
    <h:outputText value="De inhoud van de popup, bijvoorbeeld een
PanelGroup met inhoud."/>
  </f:facet>
</t:popup>
```

Afbeelding 5. De overflow getoond in een popup die wordt opgeroepen als de gebruiker een details-icon activeert.

Het eindresultaat ziet eruit als in afbeelding 5. Als we met de cursor over het Details icoontje rechtsboven bewegen, wordt de popup getoond met de aanvullende informatie over het momenteel geselecteerde boek. Door gebruik te maken van de Tomahawk component heeft deze functionele aanpassing minder gekost dan pakweg tien minuten. De resources bij dit artikel omvatten ook deze werkende ADF Faces applicatie met Tomahawk toevoeging.

Eigen uitbreidingen op Java Server Faces

Als het niet lukt om de gewenste componenten te vinden in één van de vele implementaties en library's, hebben we ook de mogelijkheid om zelf uitbreidingen op JSF te ontwikkelen. De Java Server Faces specificatie is ontwikkeld met nadrukkelijk de doelstelling om alle niveaus de standaard implementatie te complementeren of zelfs te vervangen. Met eenvoudige configuraties in de faces-config.xml file, de centrale stuurfile voor een JSF applicatie, kunnen op allerlei niveaus ingrepen worden gedaan.

De meest voor de hand liggende uitbreidingen die we zelf zouden kunnen ontwikkelen zijn Converters en Validators. De eerste worden gebruikt om de waarden zoals die in het Model zijn vastgelegd, in Strong Java Types zoals Date, Number of een eigen Type, volgens een eventueel opgegeven format-masker, te converteren in goed leesbare tekst, zoals de eindgebruiker die graag gepresenteerd wil zien. De Converter draagt ook zorg voor de omgekeerde weg: het omzetten van de string zoals de browser die naar de JSF applicatie stuurt in het door het Model verwachte type. De Validators doen een validatie van de in een component ingevoerde waarde, op basis van bedrijfsregels die bijvoorbeeld een maximale en minimale waarde specificeren. De Validator doet zijn werk nadat de conversie naar het Strong Type heeft plaatsgevonden. We zullen later in dit artikel zowel een Converter als een Validator ontwikkelen. Andere maatwerkaanpassingen die in JSF kunnen doen zijn onder meer: ViewHandler, PhaseListener, RenderKit en Filters.

Tenslotte, en waarschijnlijk het meest aansprekend, hebben we de mogelijkheid zelf JSF UI en Non-UI Componenten te ont-

wikkelen en distribueren. Op deze manier kunnen we gestandaardiseerde, herbruikbare en op maat gedefinieerde elementen tot onze beschikking krijgen die de ontwikkeling van onze web-applicaties sterk vereenvoudigen. Een eenmaal volgens de standaarden ontwikkelde JSF Component kunnen we op dezelfde manier inzetten als de standaard Sun RI componenten of bijvoorbeeld de componenten in de ADF Faces of MyFaces Tomahawk libraries.

Postcode Converter

In veel van de JSF applicaties die we in Nederland ontwikkelen zullen situaties voorkomen waar gebruikers een postcode moeten invoeren. De Nederlandse postcode heeft een duidelijk formaat: 9999XX. Het zou makkelijk zijn als we de beschikking zouden hebben over een Converter voor onze JSF pagina's die voor de gewone InputText component afdwingt dat een ingevoerd waarde aan dat formaat beantwoordt en ook zorgt dat een invoer met spaties en kleine letters wordt opgepoetst tot een nette weergave.

De test-applicatie waarin we de Postcode moeten invoeren ziet er uit als in Afbeelding 6.

Afbeelding 6. De test-applicatie waarin we de Postcode moeten invoeren.

De ontwikkeling van een eigen, herbruikbare Converter verloopt als volgt:

- Ontwikkel een class die de Converter interface implementeert
- Registreer deze Converter in de faces-config.xml file
- Optioneel: ontwikkel een Tag class die het mogelijk maakt de converter onder een eigen naam te gebruiken – ipv binnen de generieke <f:converter> tag, creëer een TagLibraryDescriptor file waarin deze tag is geregistreerd en importeer deze in de pagina.

Deze converter kan gebruikt worden met iedere JSF implementatie, van ADF Faces tot Backbase JSF Editie.

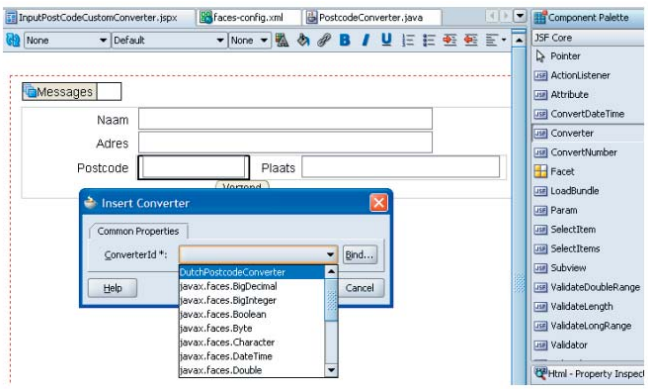
De class voor onze converter ziet er als volgt uit:

```
public class PostcodeConverter implements Converter {
    public PostcodeConverter() {
    }

    public Object getAsObject(FacesContext facesContext,
                              UIComponent uiComponent, String string)
{
        // strip spaties en maak uppercase
        String postcode = removeSpaces(string).toUpperCase();
        // match met de reguliere expressie voor de Nederlandse
Postcodes
        if (!postcode.matches("^[0-9]{4}[a-z|A-Z]{2}$")) {
            throw new ConverterException("Correct Postcode formaat is
9999XX");
        }
        return postcode;
    }

    public String getAsString(FacesContext facesContext,
                              UIComponent uiComponent, Object object)
{
        return object.toString();
    }

    public static String removeSpaces(String s) {
      StringTokenizer st = new StringTokenizer(s," ",false);
      String t="";
      while (st.hasMoreElements()) t += st.nextElement();
      return t;
      }
}
```

De converter wordt geregistreerd in de faces-config.xml file:

```
<converter>
   <converter-id>DutchPostcodeConverter</converter-id>
   <converter-class>nl.amis.jsf.converters.PostcodeConverter</converter
class>
</converter>
```

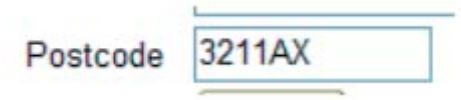
Nu kunnen we de converter toepassen door de algemene Converter component als kind van de Postcode InputText op te nemen en het converter-id van onze converter aan te geven (zie afbeelding 7).



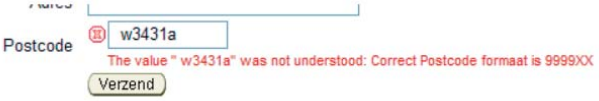
Afbeelding 7. Toepassing van de converter.

```
<af:inputText id="Postcode" columns="10">
  <f:converter converterId="DutchPostcodeConverter"/>
</af:inputText>
```

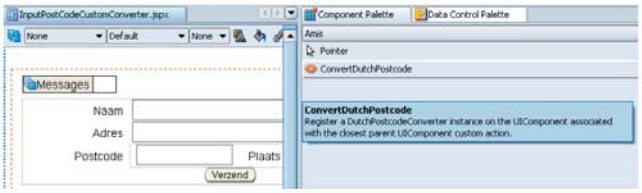
Als we nu de applicatie uitvoeren kunnen we onze Converter in actie zien. We voeren in " 3 21 1 a x" en als de Converter na submit van de waarden zijn werk heeft gedaan, zien we:



Als we daarentegen invoeren: " w3431a" zien we:



Als we een specifieke JSP Tag ontwikkelen voor de converter (zie voor de volledige code de resources link aan het eind van het artikel), wordt de Converter zichtbaar in het Component Palette en kunnen we deze naar het betreffende veld in de pagina slepen.

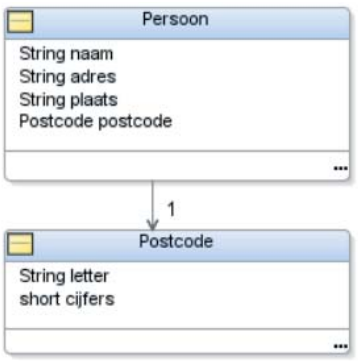


Afbeelding 8. De Converter wordt zichtbaar in het Component Palette.

De code voor het Postcode-veld wordt dan nog iets eleganter:

```
<af:inputText columns="10">
  <amis:convertDutchPostcode />
</af:inputText>
```

Tot dusverre heeft de Converter geconverteerd van String naar (nettere) String. We gaan nu een stap verder door voor de postcode een apart type te definiëren dat vanuit de class Persoon wordt gebruikt:



Afbeelding 9. Voor de postcode wordt een apart type gedefiniëerd dat vanuit de class Persoon wordt gebruikt.

In de faces-config.xml file definiëren we een managed bean onder de naam Persoon en gebaseerd op de Persoon Class. De InputText voor Postcode in onze pagina ziet er als volgt uit:

```
<af:inputText id="Postcode" columns="10"
              value="#{Persoon.postcode}">
  <amis:convertDutchPostcodeStrong/>
</af:inputText>
```

De Converter past zich aan, aan het gebruik van dit Strong Type voor Postcodes, zoals we zien in de getAsString() en getAsObject() methodes:

```
    public Object getAsObject(FacesContext facesContext,
                              UIComponent uiComponent, String string)
{
        // strip spaties en maak uppercase
        String postcode = removeSpaces(string).toUpperCase();
        // match met de reguliere expressie voor de Nederlandse
Postcodes
        if (!postcode.matches("^[0-9]{4}[a-z|A-Z]{2}$")) {
            throw new ConverterException("Correct Postcode formaat is
9999XX");
        }
        Postcode pc = new Postcode();
        pc.setCijfers(Short.parseShort(postcode.substring(0,4)));
        pc.setLetter(postcode.substring(4));
        return pc;
    }

    public String getAsString(FacesContext facesContext,
                              UIComponent uiComponent, Object object)
{
        return Short.toString(((Postcode)object).getCijfers())+((Postco
de)object).getLetter();
    }
```

Nu komt de Converter nog meer tot zijn recht: de String die de gebruiker invoert wordt door de Converter omgezet in een Strong Type waarmee het Postcode property van de Persoon Bean wordt gevuld. Andersom wordt door de Converter de Postcode weer omgezet in een net leesbare string. Als we graag een spatie zouden willen zien tussen de cijfers en de letters is het buitengewoon eenvoudig die aanpassing door te voeren in de getAsString methode van hierboven.

Postcode Validator

De functies van een Validator en een Converter sluiten vaak nauw op elkaar aan. Om een conversie succesvol te kunnen uitvoeren worden al diverse eisen gesteld aan de waarde die is ingevoerd, zoals we in de vorige alinea hebben gezien. Nu we de Postcode in een Strong Type tot onze beschikking hebben, kunnen we een validatie gaan uitvoeren. Deze zou kunnen kijken naar simpele zaken als: het eerste cijfer mag geen 0 zijn en de lettercombinatie IQ mag niet voorkomen, maar kan ook een achterliggende service raadplegen die met behulp van een post-

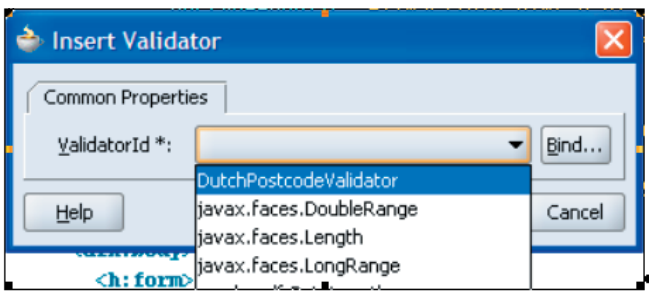
codetabel kan controleren of een bestaande postcode is ingevoerd. Een Validator maken we door een Class te ontwikkelen – en te registreren – die de Validator interface implementeert. Deze kent een enkele methode: validate. Onze PostcodeValidator zou er als volgt kunnen uitzien:

```
public class PostcodeValidator implements Validator{
    public PostcodeValidator() {
    }

    public void validate( FacesContext context
                        , UIComponent component
                        , Object value) throws ValidatorException {
        Postcode pc = (Postcode)value;
        if (pc.getCijfers()<1000 ) {
            throw new ValidatorException( new FacesMessage("Sorry, de
postcode kan niet met 0 beginnen (mag niet kleiner zijn dan 1000)."));
        }
        else if ("IQ".equalsIgnoreCase(pc.getLetter() )) {
            throw new ValidatorException( new FacesMessage("IQ is niet
een geldige lettercombinatie voor postcodes."));
        }
        // en hier de aanroep van de postcode-validatie-service....
    }
}
```

Registratie in de faces-config.xml file is nodig, net als bij de Converter:

```
<validator>
  <description>Validates that a value is in fact a valid Dutch
Postcode</description>
  <validator-id>DutchPostcodeValidator</validator-id>
  <validator-class>nl.amis.jsf.validators.PostcodeValidator</validator-
class>
</validator>
```

Nu kunnen we de standaard JSF Validator Component naar onze InputText slepen. Vervolgens kunnen we de Validator die we zojuist geregistreerd hebben selecteren:



Afbeelding 10. Selectie van de Validator die zojuist geregistreerd is.

De JSF code ziet er als volgt uit:

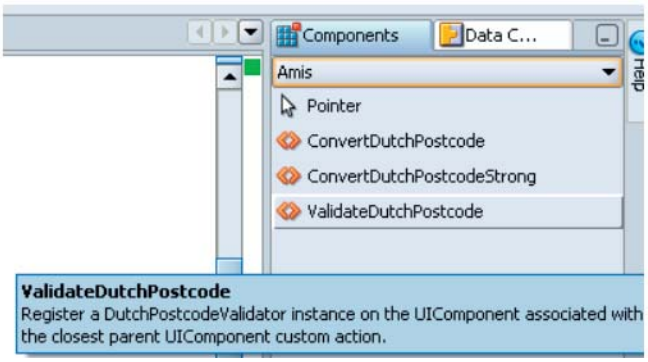
```
<af:inputText id="Postcode" columns="10"
              value="#{Persoon.postcode}">
   <amis:convertDutchPostcodeStrong/>
   <f:validator validatorId="DutchPostcodeValidator"/>
</af:inputText>
```

Als we de pagina uitvoeren en een foutieve postcode invoeren, krijgen we van de validator om onze oren:

Uiteraard kunnen we ook voor deze validator een tagclass ontwikkelen en registreren zodat we de code kunnen vereenvoudigen tot:

```
<af:inputText id="Postcode" columns="10"
              value="#{Persoon.postcode}">
  <amis:convertDutchPostcodeStrong/>
  <amis:validateDutchPostcode/>
</af:inputText>
```

en de validator gewoon van het component palette kunnen slepen:



Afbeelding 11. We kunnen voor deze validator ook een tagclass ontwikkelen en registreren.

Ontwikkeling van de InputPostcode Component

Hoewel we al waardevolle ondersteuning hebben toegevoegd voor het omgaan met postcodes in een gewone InputText component, kunnen we nog verder gaan. Het zou best aardig zijn als we in plaats van een InputText waar we zelf nog een Converter en een Validator aan moeten toevoegen, gebruik kunnen maken van een InputPostcode component die niet alleen die conversie en validatie al in zich heeft, maar bovendien precies zes posities breed is en bovendien in JavaScript afdwingt dat er vier cijfers gevolgd door twee (hoofd)letter kunnen worden ingevoerd. We zullen nu kijken naar de ontwikkeling van een custom UIComponent.

Voor een eigen JSF UIComponent hebben we tenminste nodig:
- De component class – subclass van UIComponent
- De component tag-class – voor implementatie van een JSP tag

- Registratie van de tag in een TLD file
- Registratie van de component in de faces-config.xml

Als we het netjes doen zouden we ook een Render class moeten ontwikkelen voor deze Component, bijvoorbeeld om voor verschillende RenderKits – zoals HTML, WML, XUL,… - implementaties aan te bieden. In het kader van dit artikel houden we het eenvoudig, zonder Renderer.
De InputPostcode component class ziet er als volgt uit:

```
package nl.amis.jsf.uicomponents;

// import statements weggelaten

public class InputPostcode extends UIInput {

    public InputPostcode() {
        setRendererType(null);
        this.addValidator(new PostcodeValidator());
    }

    public String getFamily() {
        return null;
    }


    public void encodeEnd(FacesContext context) throws IOException {
        // note: not in encodeBegin as the submitted value may nog
have been processed
        String clientId = getClientId(context);
        ResponseWriter writer = context.getResponseWriter();
        writer.startElement("input", this);
        writer.writeAttribute("type", "text", null);
        writer.writeAttribute("name", clientId, "clientId");
        writer.writeAttribute("size", "6", "size");
        writer.writeAttribute("maxlength", "6", "maxlength");

      if (getValue() != null) {
           writer.writeAttribute("value",
                                 new PostcodeConverterStrong().
getAsString(context,

           this,

           getValue()),
                                 "value");
        }

        writer.endElement("input");

    }

    public void decode(FacesContext context) {
        Map requestMap = context.getExternalContext().getRequest
                                                 ParameterMap();
        String clientId = getClientId(context);
        String submittedPostcode = ((String)requestMap.get(clientId));
        setSubmittedValue(new PostcodeConverterStrong().
getAsObject(context,

   this,

   submittedPostcode));

    }
```

Deze class implementeert een eenvoudige versie van InputPostcode component – nog zonder de ondersteuning voor JavaScript logica die wel in de te downloaden resources kan worden gevonden. De encodeEnd() methode schrijft het HTML input element, met lengte 6 en als waarde de door de ingebouwde converter uit het Postcode type tot String geconverteerde string. De decode() methode leest uit het request object de door de gebruiker ingevulde waarde in het Postcode veld – door een request parameter met een naam gelijk aan het Client Id op te vragen. De bijbehorende Tag class ziet er als volgt uit:

```
package nl.amis.jsf.uicomponents.tags;

//imports weggelaten

public class InputPostcodeTag extends UIComponentTag{
    private String value;

    public InputPostcodeTag() {
    }

    public String getComponentType() {
        return "InputPostcode";
    }

    public String getRendererType() {
        return null;
    }

    protected void setProperties(UIComponent component) {
        super.setProperties(component);
        if (value!=null) {
            if (isValueReference(value)) {
                FacesContext context = FacesContext.getCurrent
                                                    Instance();
                Application app = context.getApplication();
                ValueBinding vb = app.createValueBinding(value);
                component.setValueBinding("value",vb);

            }
            else {
                component.getAttributes().put("value",value);
            }
        }
    }

    public void setValue(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}
```

De tag voor de InputPostcode component moet worden geregistreerd in een Tag Library Descriptor, als volgt:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```
<taglib xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://
java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
        version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee">
  <!-- ========== Tag Library Description Elements ====================
===== -->
  <description>
    The CustomJavaServer Faces components developed by AMIS.
  </description>
  <tlib-version>1.0</tlib-version>
  <short-name>amis</short-name>
  <uri>http://nl.amis.jsf/custom</uri>
  <tag>
    <description>
      Register a PostcodeInput instance.
    </description>
    <name>inputPostcode</name>
    <tag-class>nl.amis.jsf.uicomponents.tags.InputPostcodeTag</tag-
class>
    <tei-class>com.sun.faces.taglib.FacesTagExtraInfo</tei-class>
    <body-content>empty</body-content>
    <attribute>
      <name>id</name>
    </attribute>
    <attribute>
      <name>rendered</name>
    </attribute>
    <attribute>
      <name>binding</name>
    </attribute>
    <attribute>
      <name>value</name>
    </attribute>
  </tag>
</taglib>
```

De laatste stap is de registratie van de custom InputPostcode component in de faces-config.xml file:

```
  <component>
    <component-type>InputPostcode</component-type>
    <component-class>nl.amis.jsf.uicomponents.InputPostcode</component-
class>
  </component>
```

We kunnen nu de InputPostcode component als volgt gebruiken in onze JSF pagina's:

```
  <amis:inputPostcode  value="#{Persoon.postcode}"/>
```

De component voegt zelf de Converter, de Validator en properties als lengte toe. Een meer geavanceerde implementatie van de Component zou de InputPostcode kunnen renderen als twee aparte velden – een voor de vier cijfers en een tweede voor de lettercombinatie – en gebruikmaken van JavaScript om cijfers en hoofdletters af te dwingen. De JSF pagina hoeft niet aangepast te worden om deze functionaliteit te gebruiken: die zit volledig ingebouwd in de component. Download de resources om deze geavanceerde component te bekijken en eventueel te gebruiken.

Package & Deploy

Custom componenten kunnen gebundeld worden in een JAR file die eenvoudig gedistribueerd kan worden en gebruikt kan worden in andere webapplicaties. In het JDeveloper project waarin de custom componenten zijn ontwikkeld, kunnen we een Deployment Profile aanmaken voor een JAR file. In dit profile kunnen we definiëren dat alle classes, de tld file, de faces-config.xml en de JavaScript library moeten worden gebundeld. Het uitvoeren van het deployment profile levert de jar-file op die we kunnen gaan distribueren.
Kopieer de JAR-file bijvoorbeeld naar de WEB_INF\lib directory van het project waar we de custom JSF componenten willen toepassen. Voeg de JAR file toe als Library aan het Project. Voeg de Tag Library die in de JAR file wordt gedefinieerd toe aan het project. Nu zijn de JSF componenten die in de JAR file worden gedistribueerd beschikbaar binnen het project. Zonder dat de ontwikkelaar van dat project iets hoeft te weten over Postcodes, Validaties en zelfs JavaScript heeft zij de beschikking over eenvoudig toe te passen componenten die dat wel in zich hebben.

Conclusie

De kracht van Java Server Faces is deels gelegen in de mogelijkheid componenten van verschillende aanbieders te combineren binnen één applicatie. Daar kan je dan ook nog je eigen componenten ontwikkelen en toepassen, zowel voor validatie en conversie als voor echte User Interface elementen, compleet met client-zijde JavaScript acties. Hoewel niet triviaal is de ontwikkeling van JSF componenten een duidelijk beschreven pad, dat uitkomst biedt als in geen van de tientallen compontenbibliotheken een bruikbare JSF component wordt aangeboden.
Het selecteren van JSF componenten (-bibliotheken) moet zorgvuldig gebeuren: het blijkt dat verschillende implementaties, met name degene die eigen render-kits vereisen, bijvoorbeeld voor de afhandeling van AJAX-achtige interacties, slecht te combineren zijn met elkaar. Zo kunnen ICEFaces, MyFaces Trinidad, Backbase en ADF Faces in elk geval niet gecombineerd worden binnen dezelfde applicatie. De keuze voor de primaire JSF implementatie is dus een erg belangrijke. Gelukkig zijn er nog verschillende JSF componenten sets beschikbaar die wel goed verweven kunnen worden, zoals MyFaces Tomahawk en JSF-Comp.

Lucas Jellema is Technology Manager bij AMIS. Daarnaast schrijft hij artikelen op de AMIS Weblog (http://technology.amis.nl/blog) en in tijdschriften als Optimize, Java Magazine en Java Developer Journal. Hij is zowel Oracle ACE als Oracle Regional Director for Fusion Middleware. E-mail: jellema@amis.nl.